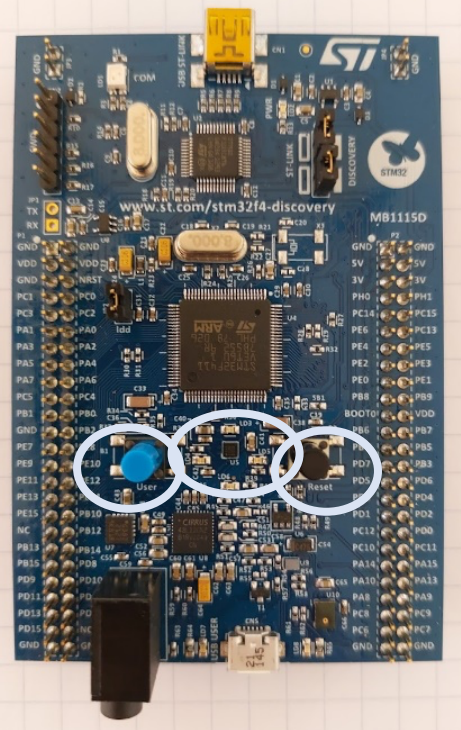
Laboratory 1

Becoming Familiar with STM32CubeIDE and the STM32F411 DISCO

1. PREREQUISITES

➢ *Use this margin to make notes!*

- You have the ENCM 515 lab kit (the STM32F411 DISCO board, a USB A to mini-B cable)
- You have successfully installed the STM32CubeIDE (see the official installation guide[1])
- You have downloaded and skimmed through the user manual for the Discovery Kit[2]
- You should download the STM32CubeF4 package before the lab, if possible[3]
- Familiarize yourself with a few elements of the board (hint: check the user manual):



LEDs

LD3 – colour: ORANGE – I/O Port: PD13

LD4 – colour: _____– I/O Port: _____

LD5 – colour: _____– I/O Port: _____

LD6 – colour: _____– I/O Port: _____

Push Button

B1 – I/O Port: _____

B2 – RESET, I/O Port: NRST

2. OUTLINE

In this lab, the aim is to become familiar with the STM32CubeIDE development environment and the STM32F411 Discovery Kit. At the heart of the Discovery Kit is an STM32F411VE microcontroller. The microcontroller is built around an Arm Cortex-M4 processor design. We will use this microcontroller throughout ENCM 515, but you should always keep in mind that the lessons you learn about using this kit should be transferrable to other development platforms. **Note that the following screenshots are from a version of STM32CUBE IDE that is not the latest release, so adjust accordingly.**

By the end of this lab, you should be:

- Comfortable with creating projects
- Navigating the various source files in a project
- Programming and debugging software for the microcontroller

[1] https://www.st.com/resource/en/user_manual/dm00603964-stm32cubeide-installation-guide-stmicroelectronics.pdf
[2] See D2L or directly: https://www.st.com/resource/en/user_manual/um1842-discovery-kit-with-stm32f411ve-mcu-stmicroelectronics.pdf
[3] https://github.com/STMicroelectronics/STM32CubeF4 --- BTW, I've uploaded a copy of this to D2L

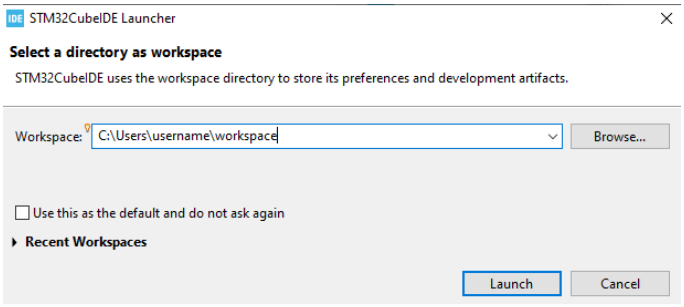
Q: There are several Questions throughout the lab for you to answer. You can collate your answers in the "Lab sheet" Word document provided (or equivalent).

PART ONE: WHAT'S WHAT

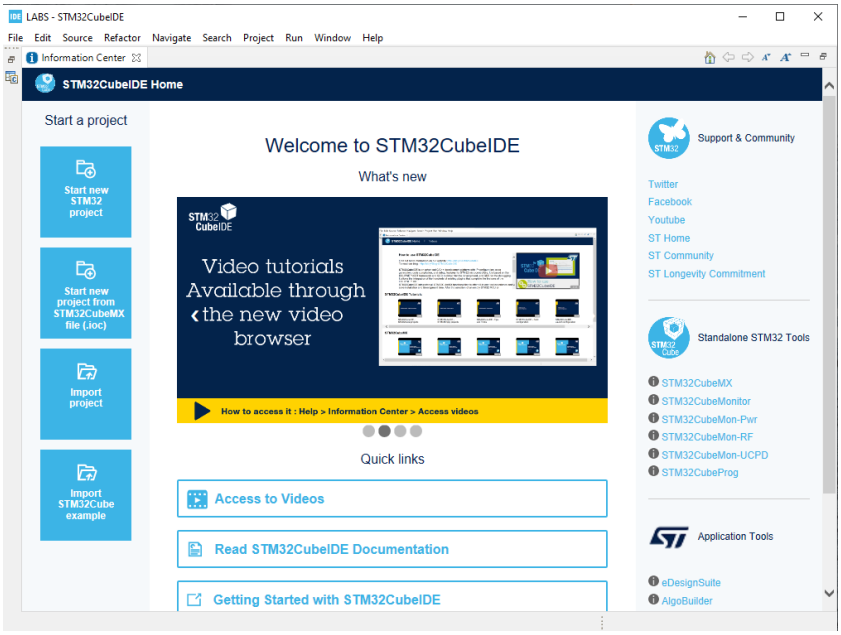
3. CREATING A PROJECT

When you start up STM32CubeIDE for the first time, you are usually presented with a dialogue box asking you to set the workspace. Set a reasonable place for this (I usually aim for a path that doesn't have any spaces). All your projects, etc. will live in the workspace.

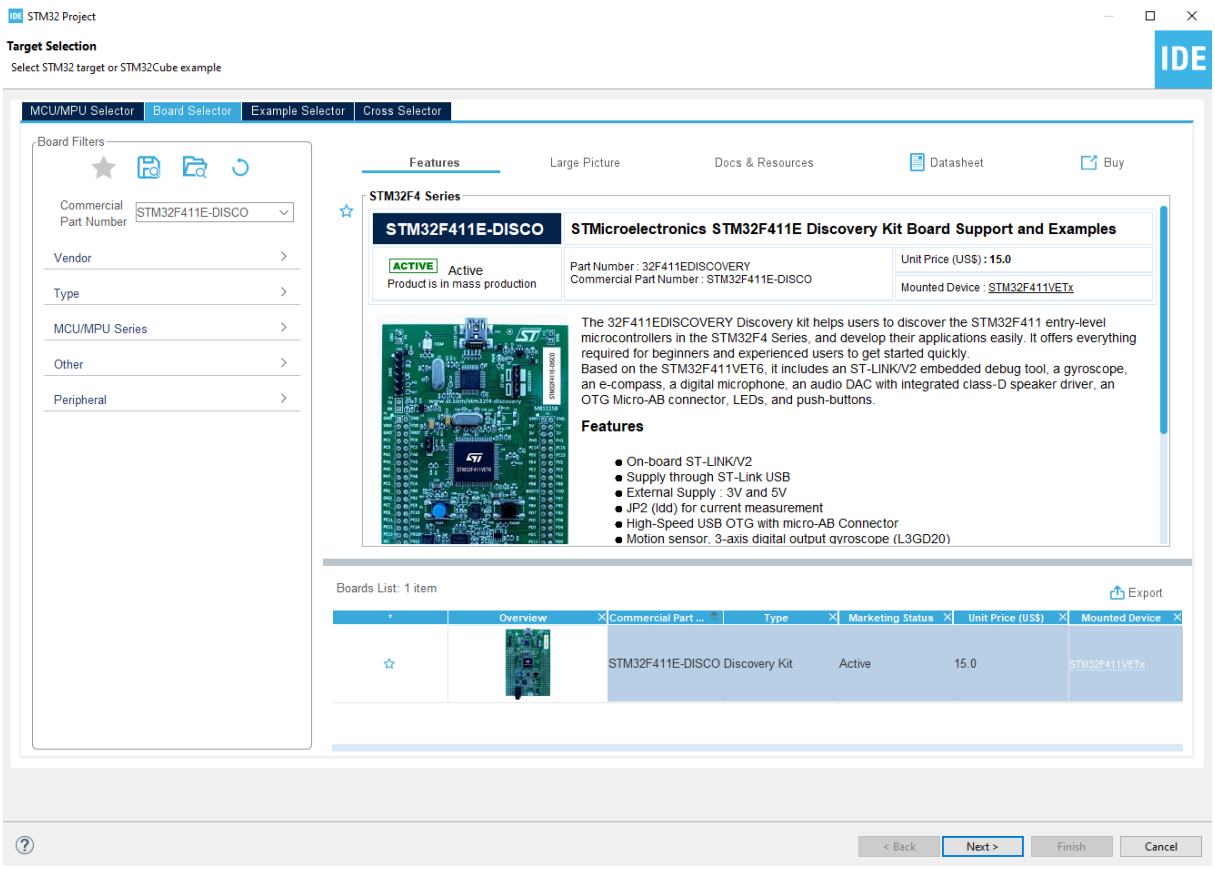
> ➢ *This should be familiar for those of you who have used Eclipse*



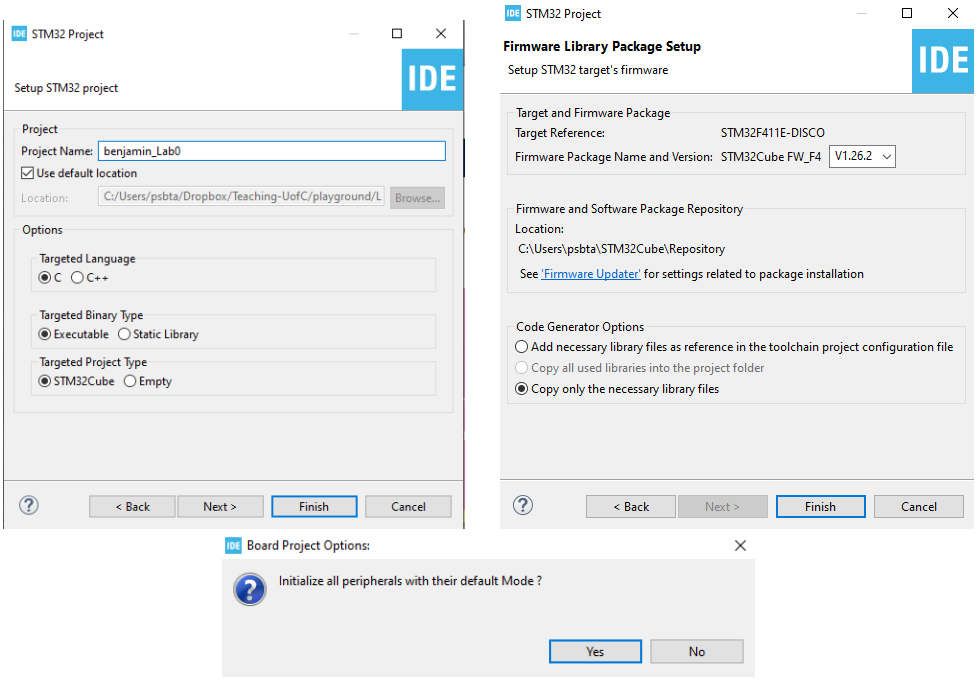
Once you have set your workspace, you'll be greeted with a welcome screen (at least for the first time; once you have projects created, it will re-open to your project).

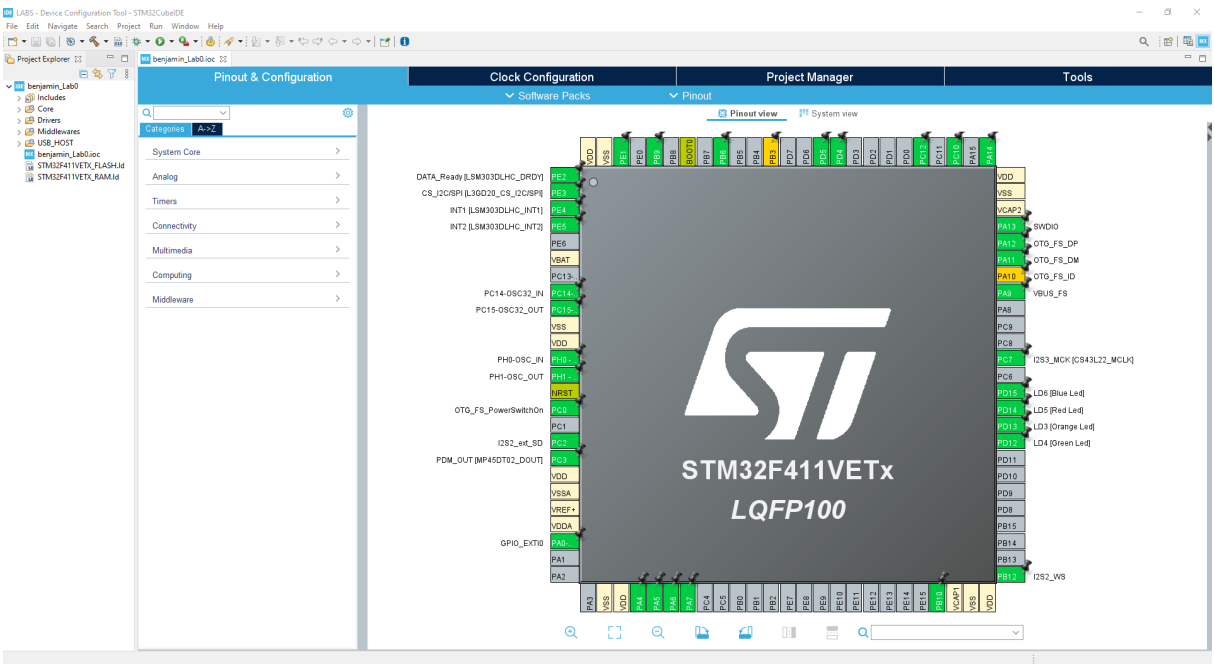


Go ahead and start a new STM32 project by clicking on the button on the right. If your machine behaves like mine, it will download some new files from the internet and then present you with a **Target Selection** dialogue box. We'll set up a project using the "Board Selector" tab at the top. You can type "STM32F411E-DISCO" in the "Commercial Part Number" part of the board selector. Take note of the other things you can explore after picking a board in the list (e.g., there's a set of handy links to the "Docs & Resources" relevant to this board).



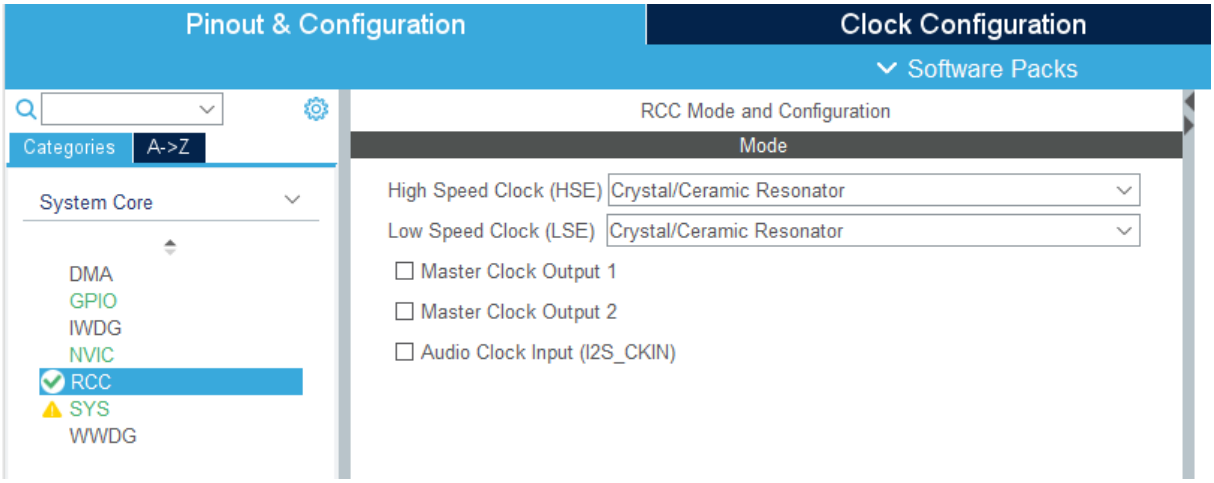
Click “Next”, set a Project Name (perhaps *yourname_Lab0*). Set the targeted language as C, the Target Binary Type as Executable, and the Target Project Type as STM32Cube. Click next and probably keep all the settings at their default. Once you click Finish, things will take a little while to get going, but eventually, you’ll be asked if you want to see the Device Configuration view. Do it. Also, if you get asked if you’d like the peripherals to be initialized to their default settings (or something to that effect), you can say yes.





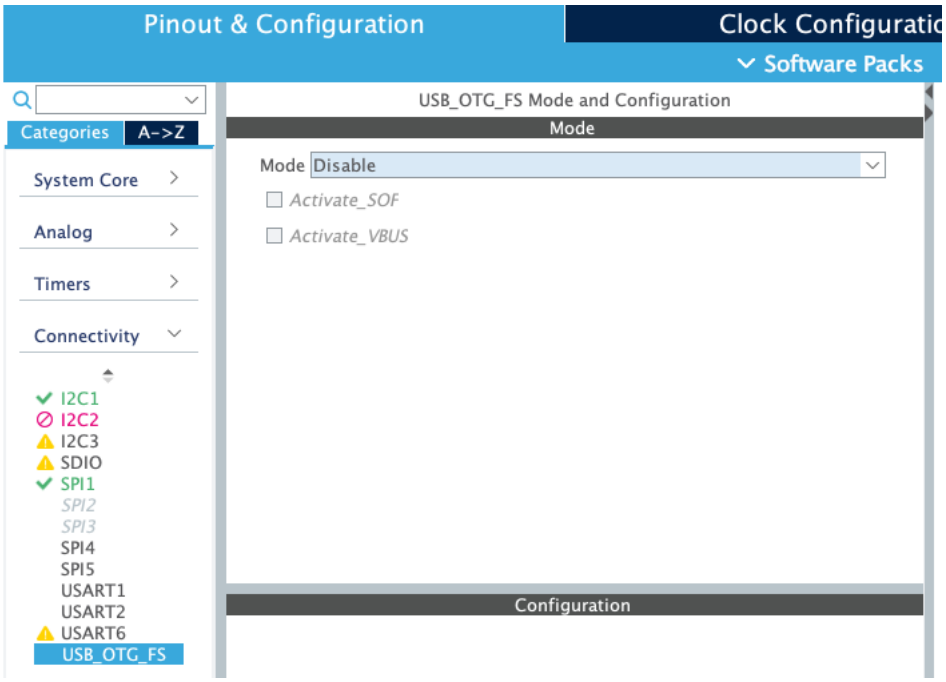
3.1. The ioc file

You should see a ".ioc" file, which you can think of as the file that holds all the project settings. Because we've started the project using a known board, a lot of the elements have been preconfigured for us (for instance, check out the PD15—PD12 pins on the right!). The Clock Configuration tab provides a nice view of the various clock frequencies that are generated for parts of the microcontroller from internally scaling the input clock signals. **For now, go back to Pinout & Configuration, select RCC, and make sure your settings are the same as below**:

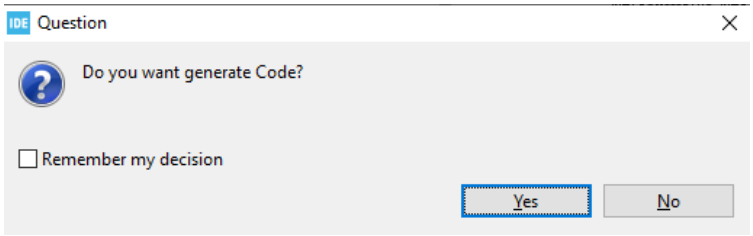


- *The HSE setting probably says something different here. You should change it to read "Crystal/Ceramic Resonator."*

There are a few extra things we don't need to use either. Click "Connectivity" and disable the USB_OTG.

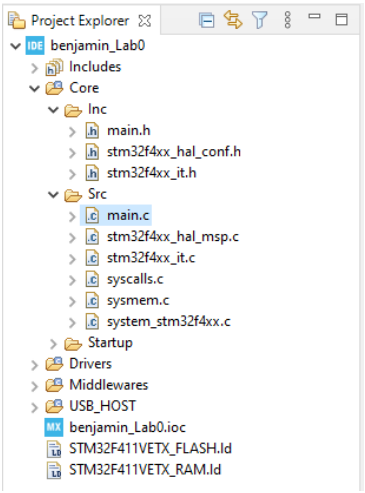


Then, when you **save**, you'll be asked to generate code. Pick Yes.



3.2. The Project Explorer

Now, let's focus on the **Project Explorer** (that appears on the left side of the window).



These various directories/sub-directories are generated by the IDE when we create our project. Almost all IDEs (at least the ones I've tried) will produce something similar when creating a new project.

Take a look at `Core > Inc > main.h` and `Core > Src > main.c`. Chances are, you will spend a lot of time trying to modify these files. These files are pre-populated with a lot of code, comments, and so on. In the STM32 workflow, a lot of code can get re-generated whenever project settings are changed. Hence, best practice is to keep the code you write contained within areas marked by the tool. For example, put your own #define statements between the `/* USER CODE BEGIN PD*/` and `/* USER CODE END PD*/` comments.

- *People like to customize their IDEs; if you ever accidentally close a part of the widow or move things around too much, you can go into Window > Perspective > Reset Perspective…*

The other files that generated include those for the **Hardware Abstraction Layer**, which provides low-level drivers and an application programming interface (API) to deal with hardware. Consider reading the STM32 documentation to get more understanding of the different firmware support that STM provides[4]. You should check out the HAL drivers in `Drivers > STM32F4xx_HAL_Driver`, such as that for `gpio`.

4. COMPILING AND RUNNING SOME CODE

4.1. Hello World for Embedded

Now, let's start to add some code! For now, we'll edit `main.c`; add the following two lines into the appropriate space of the `while(1)` loop in `int main(void)`, roughly line 117:

```
HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);

HAL_Delay(1000);
```

Note: if you start typing the code, you can use **`CTRL+Space`** to bring up the autocomplete. This can be quite helpful, especially if you can't remember exactly what the name of the function is.

After typing the function, you can hold down **`CTRL`** and **left-click** on the function name to see where the function is implemented. In this case, you'll see it is implemented in `stm32f4xx_hal_gpio.c`; the function is helpfully documented with an explanation of what the function does and the meaning of the parameters. Similar, if you go back to `main.c`, you can CTRL-click `LD3_GPIO_Port` and `LD3_Pin` to see that these are in fact helpful `#define`'s in **`main.h`** (these were generated by the IDE). You can keep trying to follow the `#define`'s (you might also need to use CTRL-F) to eventually find where the actual addresses of the various peripherals are. The idea behind using `#defines` is to make your code more readable and potentially more portable – change the platform, simply change the address(es) once.

Q1: What is the base address of the peripheral space?

Hint: You can find this by clicking through the #define or try to find this information in the appropriate documentation.

[4] https://www.st.com/resource/en/user_manual/um1730-getting-started-with-stm32cubef4-mcu-package-for-stm32f4-series-stmicroelectronics.pdf

After writing the code, let's add a breakpoint at the TogglePin(). To do this, **double-click** on the blue area by the line number of code. A small dot will appear to indicate that a breakpoint has been added.

```
115   while (1)
116   {
117       HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);
118       HAL_Delay(1000);
119     /* USER CODE END WHILE */
120     MX_USB_HOST_Process();
121
```

Now, let's **build** the project. Every time you make a change, you need to make sure that you compile the code. There are a few options to do this:

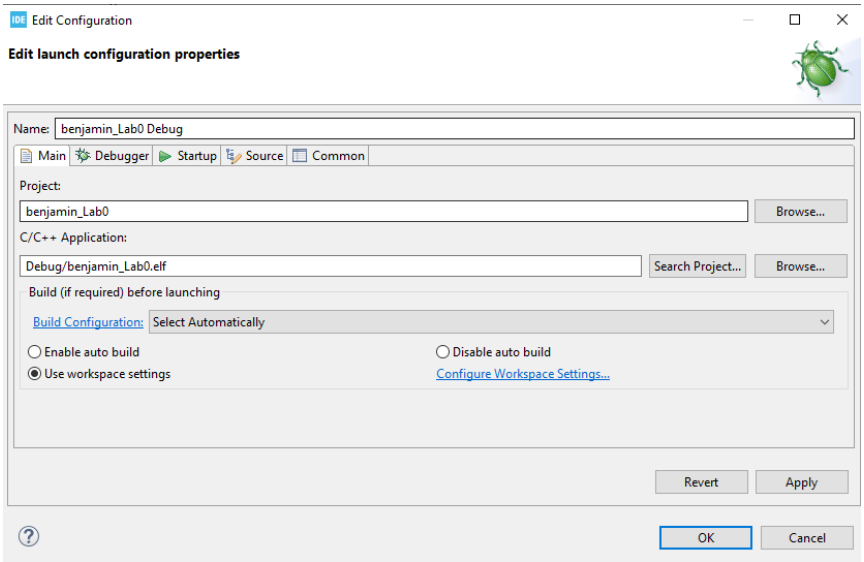
- Use the keyboard shortcut, Ctrl-B
- Right-click on the project in the Project Explorer and select Build Project
- In the menu bar at the top of the window, select Project > Build All

Once you've built the project, you should notice a few things. In the Project Explorer, there should be a Debug folder, which contains a few things. There will be something like **yourname_Lab0.elf**, which is the actual executable file produced by compiling your source code and all the drivers. The .list file shows you the disassembled code of your project. You can find the main function and the code that you've added, rendered as assembly.

4.2. Running the code (Debugging)

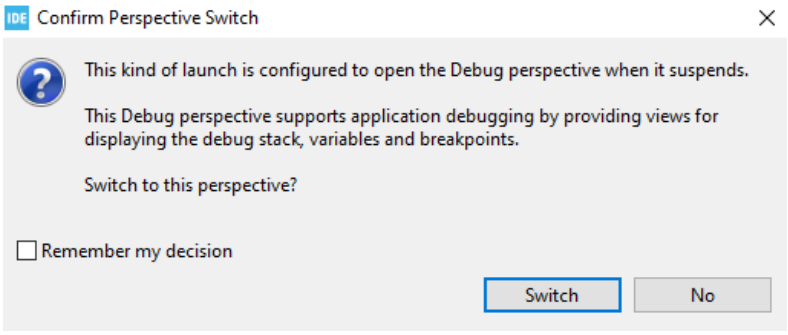
For now, let's run the code! Plug your board in, and then click the debug[5] button.

An Edit Configuration dialogue should appear:



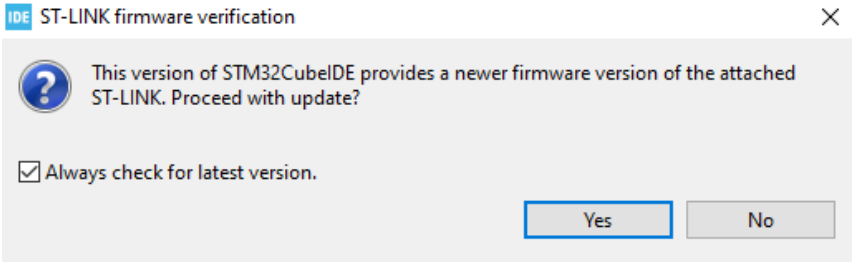
[5] You should check out the following user manual for all your debugging needs https://www.st.com/resource/en/user_manual/dm00629856-stm32cubeide-user-guide-stmicroelectronics.pdf

You can click OK without changing any of the defaults. You'll see some text scroll by in the Console, and then you'll be asked if you want to go to the "Debug Perspective" – click yes.

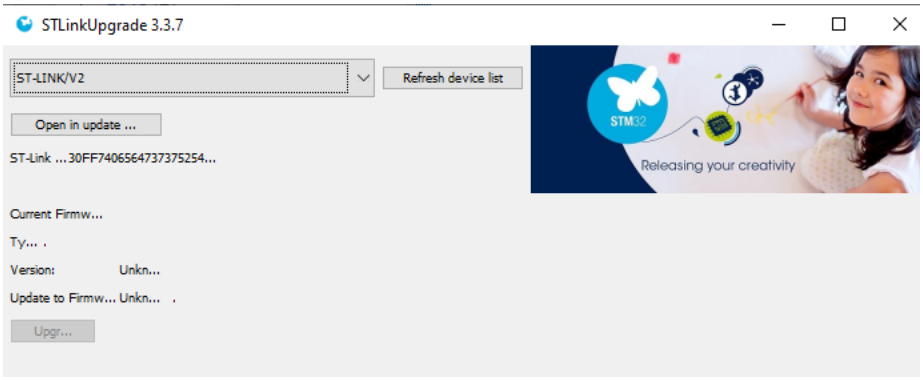


4.3. Updating the board firmware

Optional. If you are using the board for the first time, it's possible you be presented with a dialogue box asking about firmware.

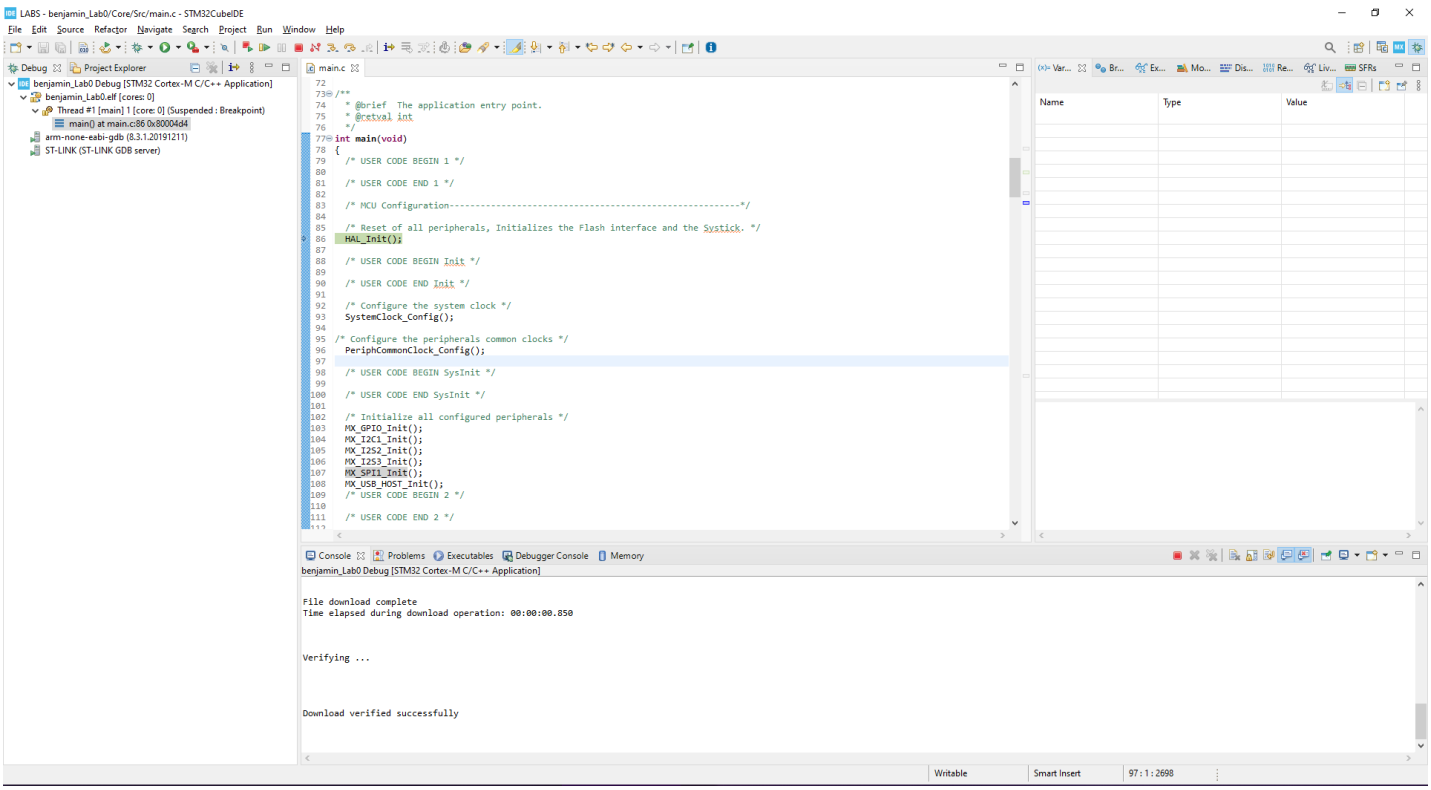


If you're feeling up to it, you can update the firmware version on your board, otherwise, just hit no and proceed. If instead you hit Yes, this will appear:



Unplug the board, then plug it back in, then click "Open in update…" Then click "upgrade". Once that's done, you can hit the Debug button again.

If all went well, you will now have the Debug perspective open, and your board will be sitting there, patiently.



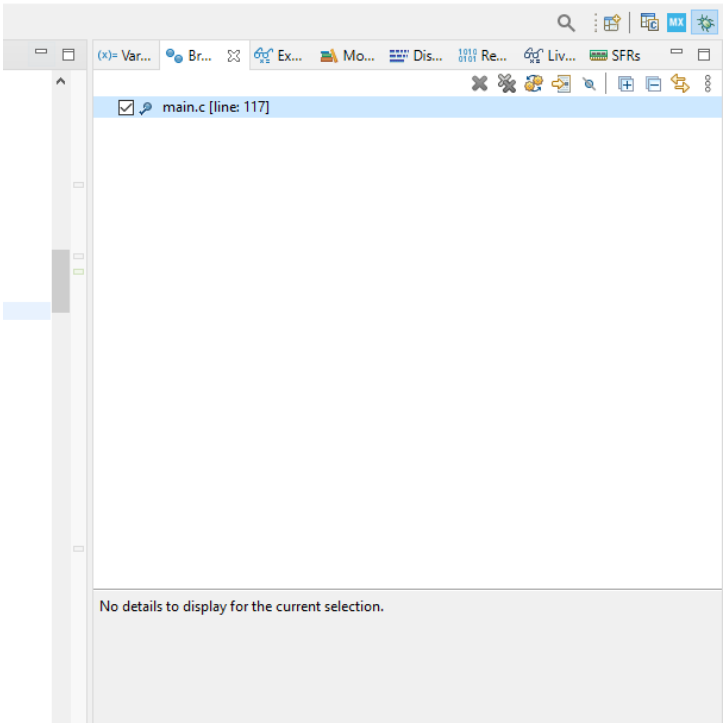
By default, the program will be paused at HAL_Init(). At this point, the Debug controls are the following buttons:

Q2: What are the functions for each of the buttons A – H?
Hint: Hover over each button to see.

Click Resume. The program should proceed until it stops again at `HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);` the processor is waiting for you to allow it to continue into the function. This is what setting breakpoints does – it marks parts of the program where the processor will "break" during debugging. Click on Resume again. One of the LEDs on your board should have turned on. Click Resume again. Does it turn off?

Now, let's disable the breakpoint. We can do this a few ways, such as by double-clicking on the region beside the line number (as we did before when setting the breakpoint). Now click Resume. Your LED should be blinking! Now add the breakpoint back.

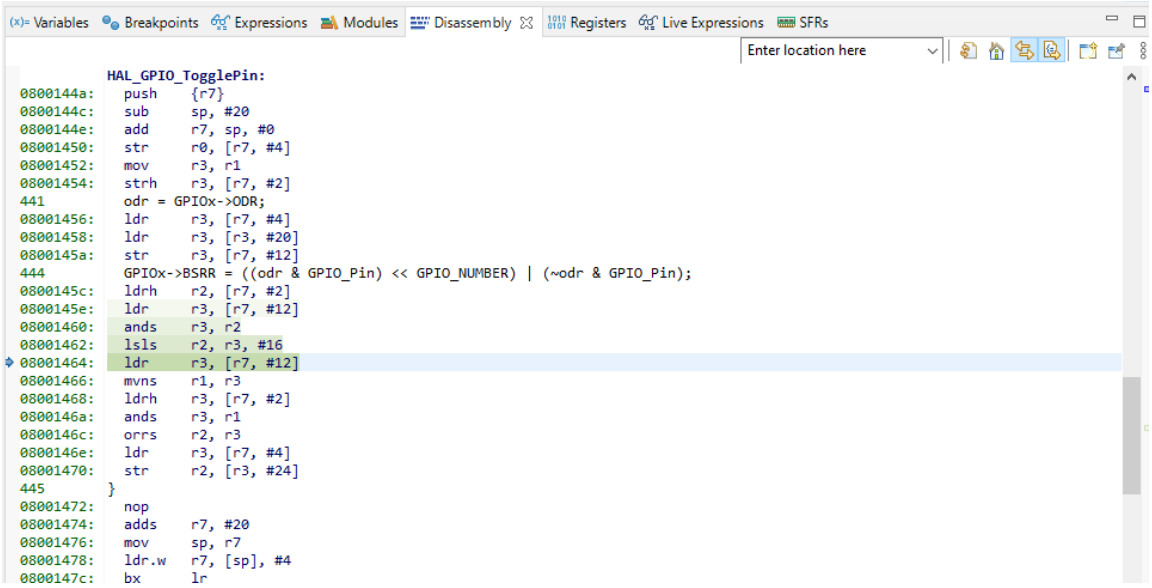
On the right side of the window, you can also click the various tabs to see things related to debugging; for instance, the second tab is actually the list of breakpoints that you have set.



From this window, you can selectively enable/disable breakpoints as well, without removing the breakpoint entirely.

As another fun thing to do, let's check out "Instruction Stepping Mode".

Using the "Step Into" we can see each line of assembly code executed one-by-one (notice the "Disassembly" view on the right side). Notice that you're also able to see the processor's internal registers updating by taking a look at the "Registers" tab.



Play around with this. Once you're satisfied, terminate debugging.

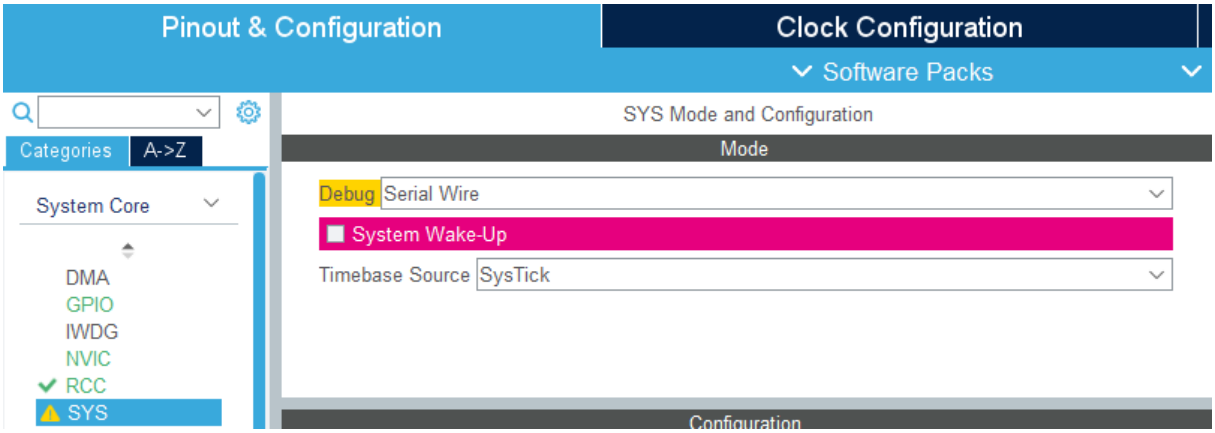
5. PRINT

5.1. Printf() 's

If you are anything like me, you've probably learned to develop software with a healthy dose of "printing to console" as a form of debugging. In embedded systems, we don't always have a screen to which we can `printf()` to! However, don't fear – processor designers have learned to include quite a lot of debugging-related hardware into their designs, which we can try to tap into to get more of an idea of what's going on inside a system that is executing our application.

In the case of Arm, they have a set of technologies called Arm CoreSight which includes a number of elements like the Serial Wire Viewer (SWV), Serial Wire Debug (SWD), Instrumentation Trace Macrocell (ITM) and Serial Wire Output (SWO). These are essentially extra pins and memories for looking into the processor. There are also other debug ports, such as the industry standard JTAG.

Let's try to use these capabilities. First, double-check that Serial Wire is set in the .ioc:



Now, let's go back to `main.c` and add a few things.

Add `#define ITM_Port32(n) (*((volatile unsigned long *)(0xE0000000+4*n)))` to the private define USER area. This defines a macro that lets us set the value of the ITM port which is located starting at 0xE0000000 (standardized by Arm in its processors). There are 32 such ports that we can use software to write to – data written to these ports can then be pulled out by the IDE. Arm suggests that we use port 0 for "printf style" debugging.

If we want to send "printf" output, we need to send **characters** (chars) byte by byte to the port. You can write this manually. Instead, we'll add a special `_write()` function that will redirect the regular "printf" to the ITM. In the `/* USER CODE BEGIN 4 */` block, add:

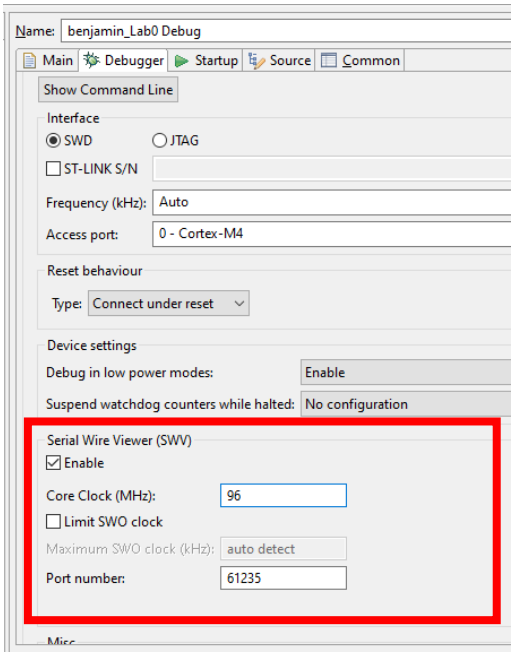
```
int _write(int file, char* ptr, int len) {
	int DataIdx;

	for (DataIdx = 0; DataIdx < len; DataIdx++) {
		ITM_SendChar(*ptr++);
	}
	return len;
}
```

Then, add a `printf()` somewhere in your code, for example, in the while(1) loop just after where you toggle the LED. **NOTE:** you must end the string that you want to print with `\n`.

Build the project.

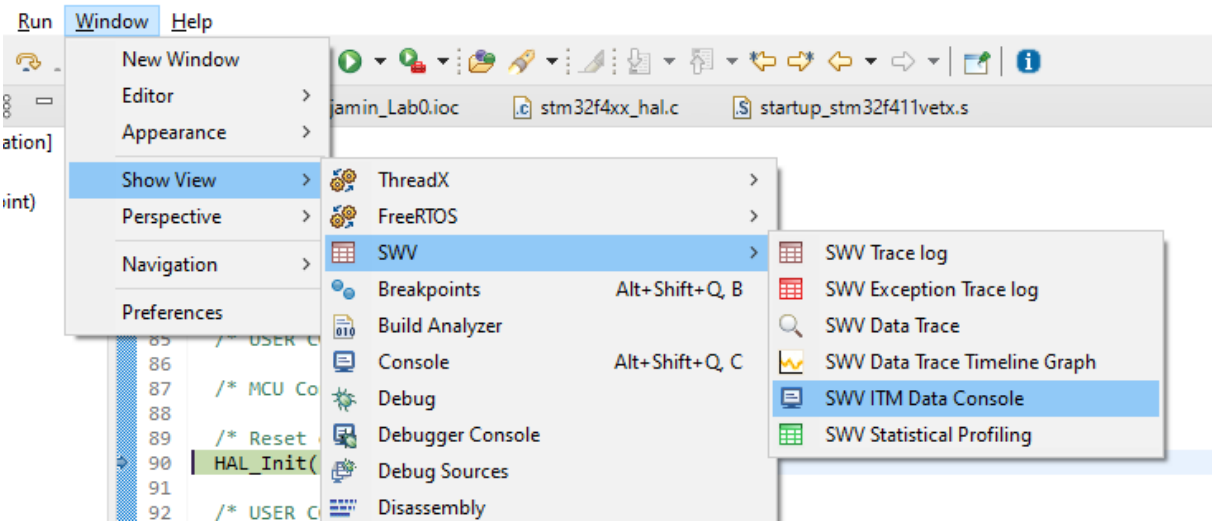
5.2. Setting up the Serial Wire Viewer

Now, before we can make use of this, we need to set up our Debug configuration to activate the Serial Wire Viewer. So, click the Debug button's dropdown and select Debug Configurations. Your debug configuration will probably be selected already, otherwise, pick it from the list on the left. Click the Debugger tab, enable Serial Wire Viewer, and enter the Core Clock frequency (which you can find out in the .ioc).

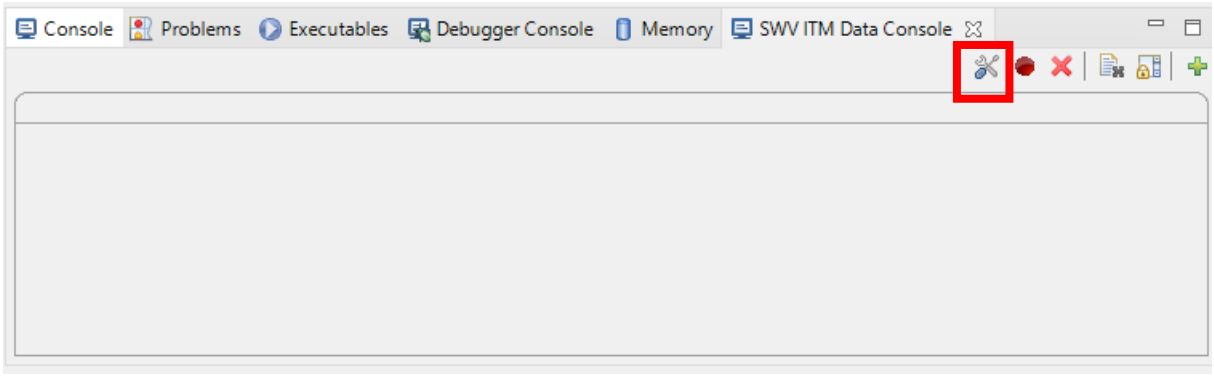


Hit apply, and then Debug.

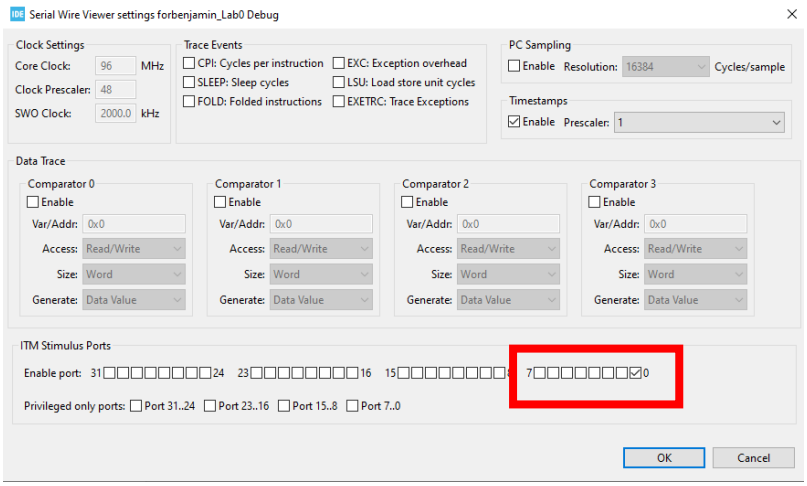
As before, your code should be downloaded to the board, and it should be waiting for you to hit Resume. Before we do this however, we need to open one of the SWV viewers. In the menu bar, go to Window > Show View > SWV > and select SWV ITM Data Console. Sometimes SWV might not appear directly in the Show View menu, so you might need to click on "Other".



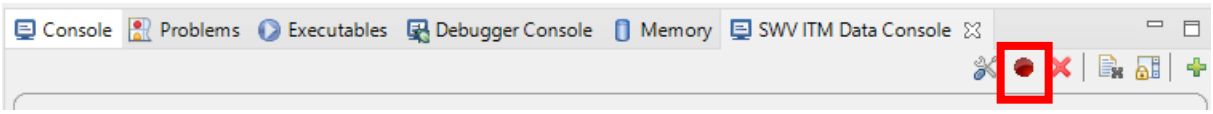
The SWV ITM Data Console appears in the bottom



Click the little wrench/screwdriver button and open up the Serial Wire Viewer settings. Enable port 0 and then hit “OK”.



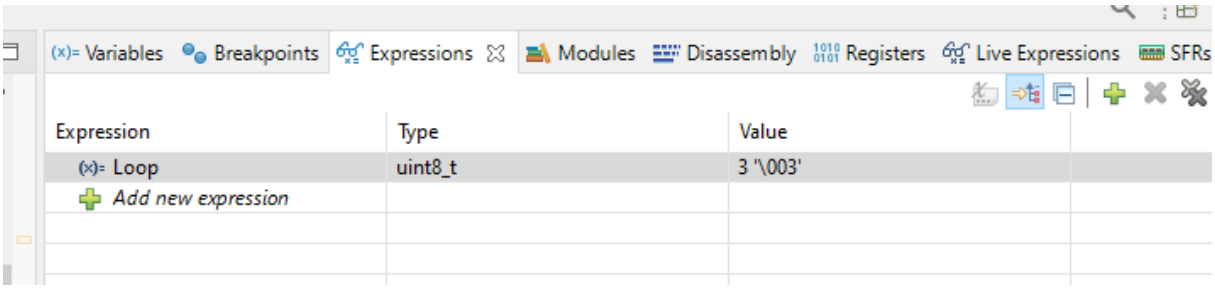
Back in the SWV ITM Data Console, select “start trace” (which is the Red dot) and then you can hit “Resume” in the debugger.



You should now see some printf() outputs in the SWV ITM Data Console!

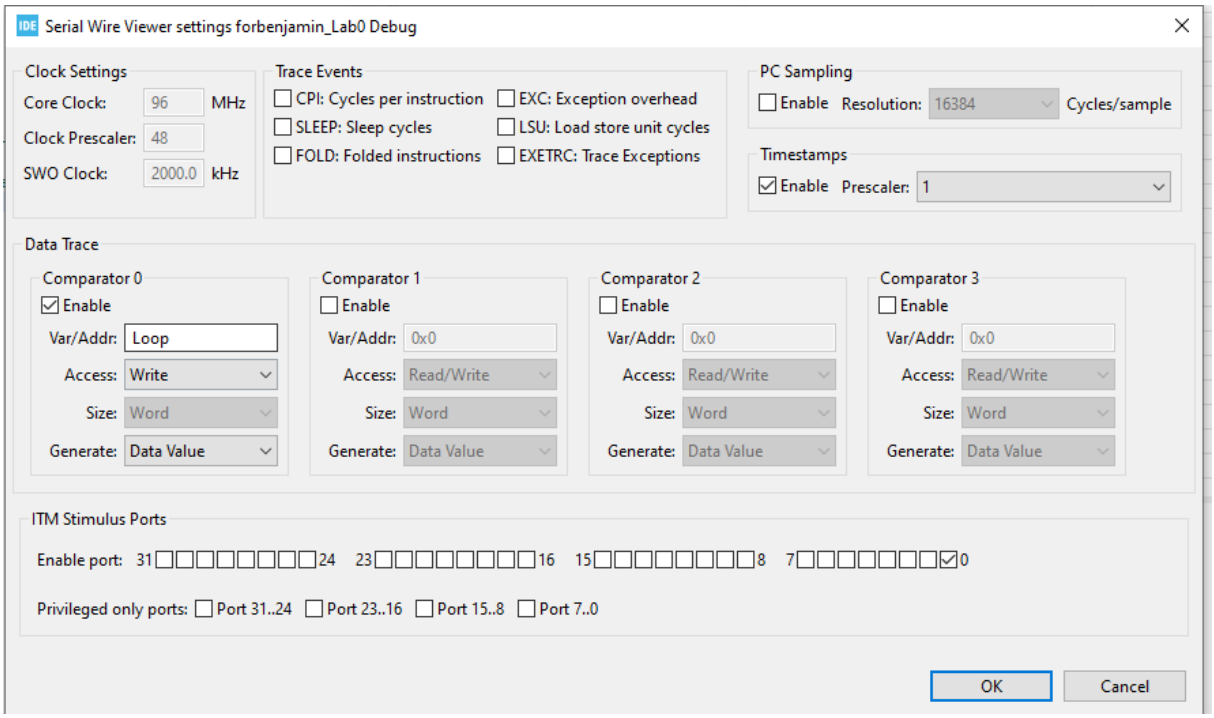
5.3. Try it yourself…
For our next trick:

- Add a global variable `uint8_t Loop = 0;` to the `/* USER CODE BEGIN 0 */` section
- In the while(1) loop, add some code to increment Loop every second, up to 100. Once Loop is 100, set it back to 0, and then continue incrementing.
- Build the project, and then debug again. This time, before debugging, select one of the places you’ve written “Loop”, right click on it, and select Add Watch Expression
- As you step through the program, you should be able to see the value of Loop change in the Expressions tab. You’ll need to set an appropriate breakpoint so that it pauses the program (so you can see the value of Loop at that instance).

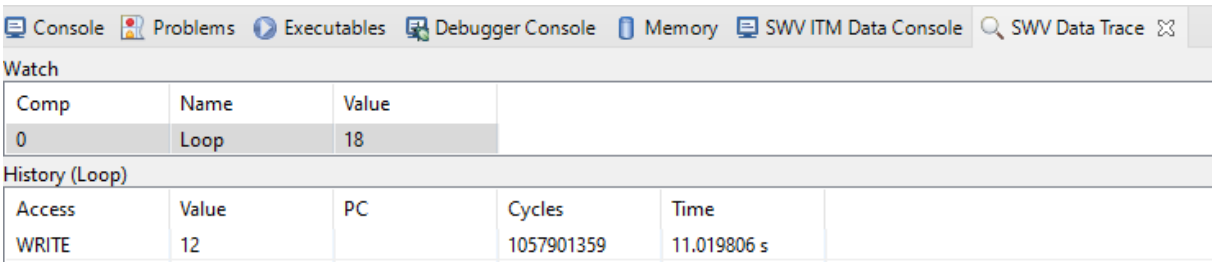


Another thing we should do is use the SWV again. “Terminate and Relaunch” the debug session. Then, before starting, let’s open up the SWV Data Trace. Hint: it should be in the same place you found the SWV ITM data console.

Change the settings to enable Comparator 0. We’ll use it to look at Loop, and specifically, any time that processor writes to that variable.



You’ll see something like this appear:



Q3: How much time is elapsed between each WRITE? Is this expected?

Another neat thing to try is using the SWV Data Trace Timeline Graph. See if you can find it and use it.

Q4: Take a screenshot of the SWV Data Trace Timeline Graph. You might need to wait some time to get an interesting shape to appear. Alternatively, adjust your delay so that the Loop value is incremented more often.

6. INLINE ASSEMBLY

Sometimes, you might find yourself really wanting to write things in assembly[6]. This is probably more likely the case when you are using a custom or slightly more esoteric architecture. Different compilers can offer different ways to mark snippets of assembly; as we're using GCC and C, we will use the `__asm` keyword. The general format is:

```
__asm [volatile] (code);
```

Where you add into the code the assembly instructions enclosed in quote marks. For example:

```
__asm volatile ("ADD R0, R1, R2");
```

The `volatile` keyword tells the compiler to leave it as is (i.e., not to optimize).

You can also use a more sophisticated approach. For a good explanation, check out Arm's documentation[7].

```
__asm [volatile] (code_template
: output_operand_list
[: input_operand_list [: clobbered_register_list]]
);
```

The nice thing about this format is it lets you mix together C and assembly more explicitly. Consider the following function:

```
int test_assembly(int a, int b) {

	int res = 0;

	__asm volatile ("ADD %[result], %[input_i], %[input_b]"
	: [result] "=r" (res)
	: [input_i] "r" (a), [input_b] "r" (b)
	);

	return res;
}
```

You can see that we specify C variables as the source/destination of the instruction. The symbolic name [result] is mapped to the C variable "res".

Add this function to your code and call it in `main()` (in the `/* USER CODE BEGIN 4 */` block), probably before the while(1) loop. Build the project, and take a look at the listing.

Q5: Copy the listing for the `test_assembly` function and annotate it. Mark which lines of assembly correspond to what you've written and what corresponds to the compiler-inserted code for the Calling Convention. Explain what's going on (e.g., here we can see the registers that will be overwritten are being saved on the stack…. Now we see them restored…)

[6] Okay, maybe that's not really that likely
[7] https://developer.arm.com/documentation/100748/0616/Using-Assembly-and-Intrinsics-in-C-or-C---Code/Writing-inline-assembly-code

PART TWO: EXPLORING ARITHMETIC OPERATIONS

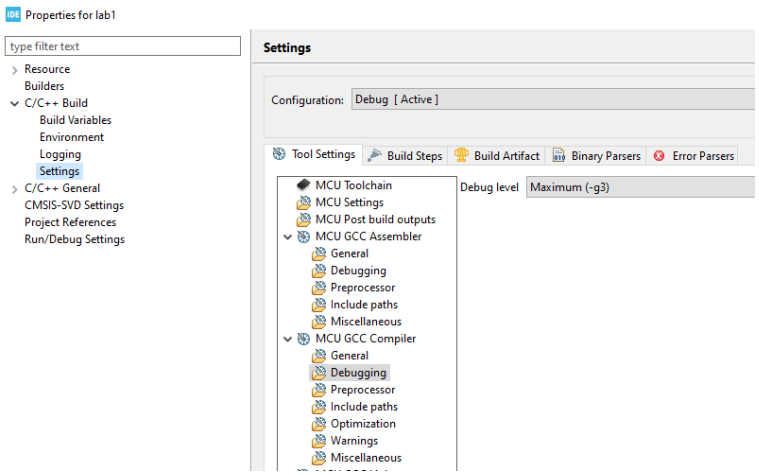
7. CREATE A NEW PROJECT (AGAIN)

Create a new STM32CubeIDE project for the STM32F411E DISCO (follow the earlier instructions as required). Don't forget to change the settings for the RCC Mode and Configuration (i.e., set the HSE to use the Crystal/Ceramic Resonator). Once your project is created, you should **add the necessary code** and **change the required settings** so that you can see the output of `printf()`'s on the SWV ITM Data console.

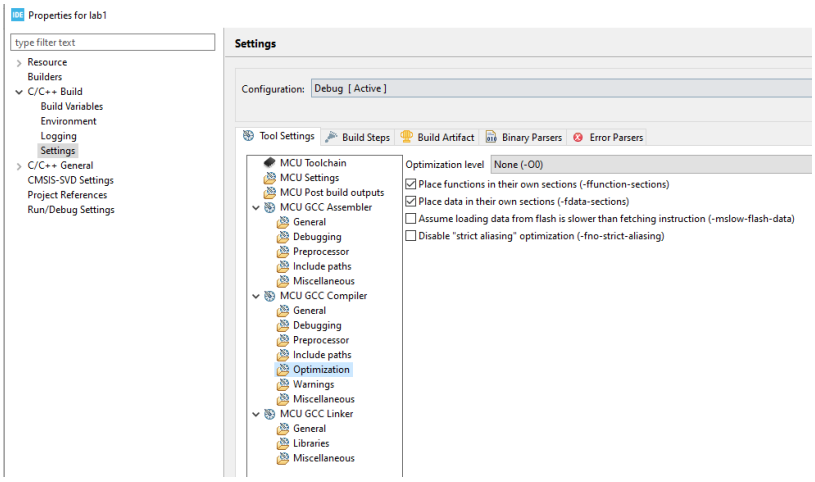
In this part of the lab session we will make use of the ITM stimulus ports, so let's add the following handy macro. Later on, this will let us write values to the n^{th} port.

```
#define ITM_Port32(n)     (*((volatile unsigned long *)(0xE0000000+4*n)))
```

Because we are using this lab to explore some of what's happening under the hood, we should make sure that we are compiling with debug information and that we are not letting the compiler do too much optimization for us at this stage. Right click on your project in the Project Explorer and select properties.



Make sure Debug level is set to Maximum.



And that optimization level is None.

Note that, in practice, you will change these settings depending on where you are in the design flow. For instance, when it's time to send your code into production, it might not be necessary to include

debug information (and in some cases, you might want to remove the debug information for intellectual property reasons). Changing the debug level can help improve code size. In main(), after the initialization stuff, add `HAL_SuspendTick();`

Q1 > Once you have set up your project, add a `printf()` to `main()` that prints your name, the date, and a short joke to the SWV ITM Data Console. Take a screenshot of the console output after running your code on the STM32F411E Disco.

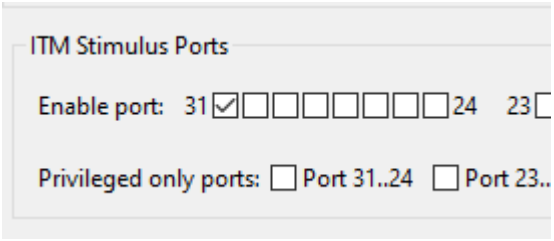
8. INSTRUMENTING CODE

Now, let's try to get a sense of how long it takes for some code to execute on this platform. **Profiling** the performance of software is an important part of any design process as it allows us to get a sense of potential bottlenecks and inefficiencies.

First, **remove** the `printf()` that you added for Q1. Write a function `void BasicLoopTest(void)` that contains a for loop that loops 2,000,000 times. Call this function in `main()` (somewhere in the `/* USER CODE BEGIN 2 */` section). Write to ITM port 31 before and after calling the function, like so:

```
ITM_Port32(31) = 1;
BasicLoopTest();
ITM_Port32(31) = 2;
```

Build and then start debugging. Before resuming, open the SWV Trace Log view, and make sure you enable port 31 in the trace configuration. Resume the program.



Take a look at the SWV Trace Log. You should notice that there are two entries for ITM Port 31.

Q2 > Make a note of the time stamps and cycles for the two ITM Port 31 entries. How many clock cycles are there between the two writes? How much time has elapsed? What is the average amount of time taken for each loop iteration?

After running it once through, restart the program and instead switch to **Instruction Stepping mode**, when the processor starts executing `BasicLoopTest`. Make a note of which instructions are being executed.

9. BASIC OPERATIONS ON FLOATING/FIXED POINT NUMBERS

To get a sense of the differences between floating point and fixed point operations on this platform, let's try out some arithmetic in C.

Add the function prototypes for two new functions, called `void FloatingExperiment(void)` and `void FixedExperiment(void)`.

9.1. Floating Point Experiments

Let's start with the following:

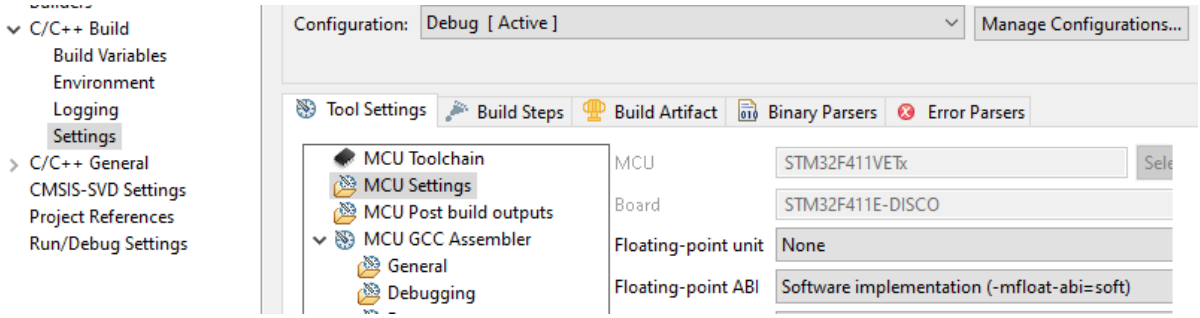
```
void FloatingExperiment(void) {
	float a = 0.5;
	float b = 0.125;
	float c = a * b;
}
```

Call this function in `main()` and surround the function call with writes to `ITM_Port32(31)` so we can get the cycle count before and the cycle count after. Start the debugger and add a breakpoint at the start of this function. Then, when you're in the function, take a look at the instructions being executed as well as the different contents of the floating point registers.

Q3 > How many clock cycles are taken to perform the floating point function? What happens if you change the value of a and b to something that is not a "clean" power of 2. Try a few different values (maybe 5), and record the time taken. Include situations where the numbers are different magnitudes (like 10-point something multiplied by 0-point something…)

Note, if you want to be able to print floats to the console (i.e., `printf("%f\n", some_float);`), you will need to add a special flag to the linker. To do this, go into the Project Properties, select C/C++ Build > Settings > Tool Settings > MCU GCC Linker > Miscellaneous, and then add `-u _printf_float` to the Other flags.

For our next trick, let's tell the compiler to pretend we don't have a floating point unit, by changing the project settings.



Q4 > How many clock cycles are taken to perform the floating point function now?

Once you've done this experiment, be sure to change the project settings back to what they were before.

9.2. Fixed Point Experiments

9.2.1. General Fixed Point

Now, let's try a similar experiment, but with Fixed Point numbers. Let's use Q1.15 format for now. Because there's no "built-in" data type for Q numbers (more on this later), we'll use int16_t.

```
void FixedExperiment(void) {
	int16_t a = ???; // should be 0.5;
	int16_t b = ???; // should be 0.125;
	int16_t c = a * b; // leave this as is initially
}
```

Q5 > Fix the code snippet so that the correct values are loaded in a and b. First, leave the multiplication as is. Debug the program and observe what's happening when you step through the instructions. You will probably get a wrong/unexpected value for c – modify the statement so that you get the correct result in the int16_t. How many cycles does this take?

9.2.2. CMSIS

By now we should be able to infer that ARM intends their designs to do DSP, and to that end, they provide a helpful library that helps us implement algorithms faster and in a more portable way. The ARM Common Microcontroller Software Interface Standard (CMSIS) is a vendor-independent abstraction layer. In this course, we will pay particular attention to the CMSIS DSP Library.[8] Let's download STMicroelectronic's version for their STM32F4 family from here if you haven't already. Once we've downloaded it, we can integrate the DSP libraries into our project using the official instructions (a PDF version is on D2L).[9] Follow the instructions up to (and including) "add the required header file "arm_math.h"", and then that's come back to this manual (i.e., no need to add the floats or FFTs that they suggest there).

Note a few things. First, including the CMSIS library[10] provides us with some handy helper types, like `q7_t, q15_t, float32_t`. It also provides a few helpful functions, some of which we'll consider in future labs. For now, let's check out:

void	**arm_float_to_q15** (const **float32_t** *pSrc, **q15_t** *pDst, uint32_t **blockSize**)
	Converts the elements of the floating-point vector to Q15 vector. More...

Create a new function, `void CMSISExperiment(void);`. In this function, you should declare three arrays: an array of 10 different floats, and empty array of q15_t, and an empty array of float32_t. Pick a few random floats in the range [-1,1) and a few outside. Call the above function with the q15_t array as the destination Using this result, call a function to go back to float.

Q6> Take a screenshot of the Variables window after calling these functions. What do you observe?

Q7> Write a reflection of what you have observed and learned in this lab

To submit:

- **Add your lab sheet and your main.c to a zip archive and upload to D2L.**

[8] https://arm-software.github.io/CMSIS_5/DSP/html/index.html
[9] Use the official instructions here: https://community.st.com/t5/stm32-mcus/how-to-integrate-cmsis-dsp-libraries-on-a-stm32-project/ta-p/666790
[10] You should do an online search to find the documentation for CMSIS-DSP